Servlet 详解

主要内容

JavaEE 简介

Servlet 简介

Tomcat 详解

Servlet 技术详解

学习目标

知识点	要求
JavaEE 简介	了解
Servlet 简介	了解
Tomcat 详解	掌握
Servlet 技术详解	掌握

一、 JavaEE 简介

1 什么是 JavaEE

JavaEE（Java Enterprise Edition），Java 企业版，是一个用于企业级 web 开发平台。最早由 Sun 公司定制并发布，后由 Oracle 负责维护。在 JavaEE 平台规范了在开发企业级 web 应用中的技术标准。

在 JavaEE 平台共包含了 13 个技术规范(随着 JavaEE 版本的变化所包含的技术点的数量会有增多)。它们分别是：JDBC、JNDI、EJB、RMI、Servlet、JSP、XML、JMS、Java IDL、JPA、JTA、JavaMail 和 JAF。

2 JavaEE 版本

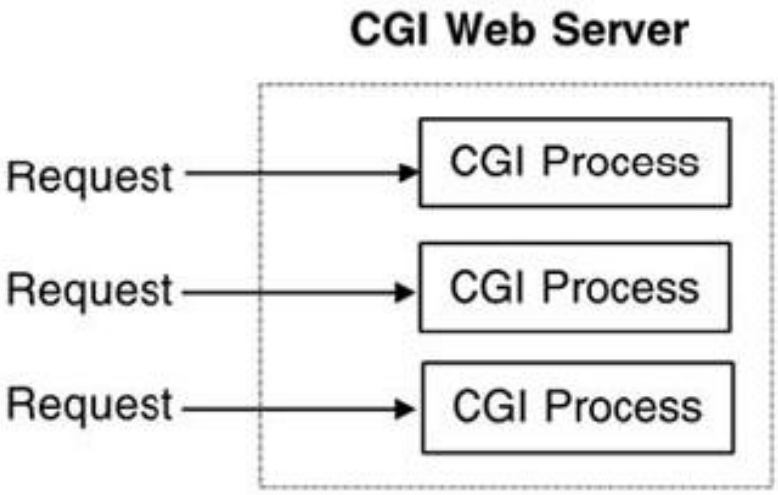
年份	版本	对 JDK 的支持	Servlet 版本	JSP 版本
1999 年 12 月	1.2	1.1 and later	2.2	1.1
2001 年 9 月	1.3	1.3 and later	2.3	1.2
2003 年 11 月	1.4	1.4 and later	2.4	2.0
2006 年 5 月	5.0	5 and later	2.5	2.1
2009 年 10 月	6.0	6 and later	3.0	2.2
2013 年 6 月	7.0	7 and later	3.1	2.3
2017 年 8 月	8.0	8 and later	4.0	2.3

二、 Servlet 简介

1 Web 开发历史回顾

1.1 CGI

公共网关接口（Common Gateway Interface，CGI）是 Web 服务器运行时外部程序的规范。



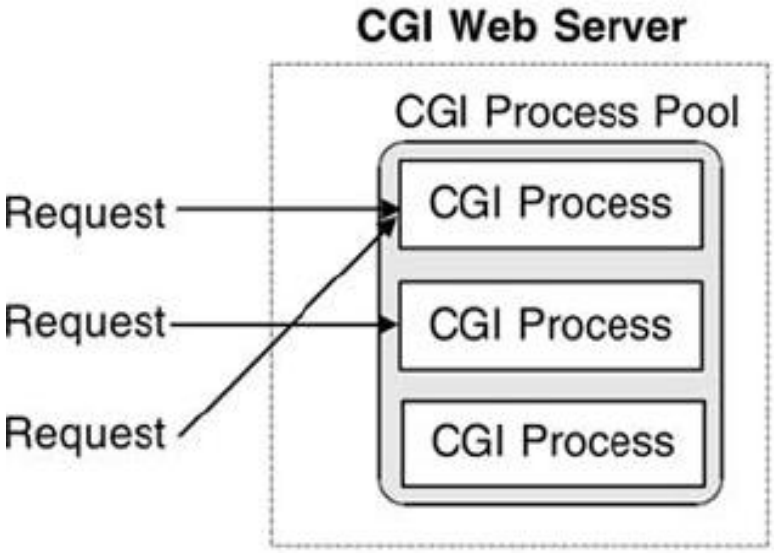
1.2 CGI 缺点

- 以进程方式运行，对每一个客户端的请求都要启动一个进程来运行程序，导致用户数目增加时，服务器端资源被大量占用。

- 由于对操作系统和进程的不深刻理解，使得开发人员开发的 CGI 程序经常遇到莫名其妙的错误。
- 不同的 CGI 之间不能共享资源

1.3 FastCGI

FastCGI 是对 CGI 模式的一个改进，采用了 Pooling 技术，一定程度上改善了性能，但是由于仍然是基于进程运行的所以并没有从根本上解决问题。

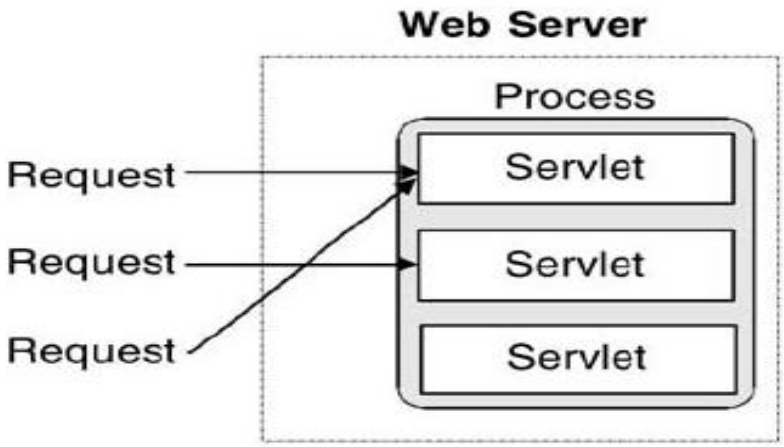


2 Servlet 介绍

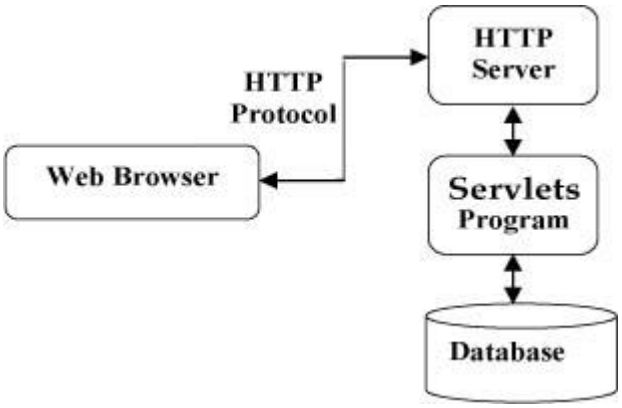
Servlet 是 Server Applet 的简称，称为服务端小程序，是 JavaEE 平台下的技术标准，基于 Java 语言编写的服务端程序。 Web 容器或应用服务器实现了 Servlet 标准所以 Servlet 需要运行在 Web 容器或应用服务器中。Servlet 主要功能在于能够在服务器中执行并生成数据。

2.1 Servlet 技术特点

Servlet 使用单进程多线程方式运行。



2.2 Servlet 在应用程序中的位置



三、 服务器

1 JavaEE 应用服务器（JavaEE Application Server）

应用服务器是 Java EE 规范的具体实现，可以执行/驱动基于 JavaEE 平台开发的 web 项目。绝大部分的应用服务器都是付费产品。

常见的应用服务：

Weblogic（BEA Oracle 收费）

Webshpere（IBM 收费）

JBoss（RedHad 收费）

Resin（Caucho 收费）

JRun（Macromedia 收费）

Geronimo（Apache 免费）

2 Web 容器（Web Server）

只实现了 JavaEE 平台下部分技术标准，如 Servlet，Jsp，JNDI，JavaMail。Web 容器是开源免费的。

Tomcat（Apache 开源免费）

Jetty（Jetty 开源免费）

3 Tomcat 的使用

3.1 Tomcat 下载与安装

3.1.1 下载

下载地址：http://tomcat.apache.org/

3.1.2 安装

Tomcat 是绿色软件，解压就可使用。

3.1.3 配置环境变量

Tomcat 是用 Java 语言开发的 Web 容器，所以在使用 Tomcat 时需要在操作系统中正确配置环境变量。

JAVA_HOME：C:\Program Files\Java\jdk1.8.0_171

PATH：%JAVA_HOME%\bin;

CLASS_PATH：%JAVA_HOME%\lib;

3.2 Tomcat 目录结构与介绍

3.2.1 bin

bin 目录主要是用来存放 tomcat 的命令文件，主要有两大类，一类是以.sh 结尾的（linux 命令），另一类是以.bat 结尾的（windows 命令）。

3.2.2 conf

conf 目录主要是用来存放 tomcat 的一些配置文件。

3.2.3 lib

lib 目录主要用来存放 tomcat 运行需要加载的 jar 包。

3.2.4 logs

logs 目录用来存放 tomcat 在运行过程中产生的日志文件。

3.2.5 temp

temp 目录用户存放 tomcat 在运行过程中产生的临时文件。（清空不会对 tomcat 运行带来影响）

3.2.6 webapps

webapps 目录用来存放应用程序，当 tomcat 启动时会去加载 webapps 目录下的应用程序。可以以文件夹、war 包的形式发布应用。

3.2.7 work

work 目录用来存放 tomcat 在运行时的编译后文件，例如 JSP 编译后的文件。

3.3 Tomcat 启动与关闭

Tomcat 的启动与关闭需要执行 bin 目录中的命令脚本。

3.3.1 Tomcat 启动

运行 startup.bat 文件。

3.3.2 Tomcat 关闭

3.3.2.1 方式一

运行 shutdown.bat 文件。

3.3.2.2 方式二

直接关闭掉启动窗口。

3.3.3 访问 Tomcat

访问 Tomcat 的 URL 格式：http://ip:port

访问本机 Tomcat 的 URL 格式：http://localhost:8080

3.4 Tomcat 的配置

3.4.1 Tomcat 配置文件介绍

Tomcat 的配置文件由 4 个 xml 组成，分别是 context.xml、web.xml、server.xml、tomcat-users.xml。每个文件都有自己的功能与配置方法。

3.4.1.1 context.xml

Context.xml 是 Tomcat 公用的环境配置。 Tomcat 服务器会定时去扫描这个文件。一旦发现文件被修改（时间戳改变了），就会自动重新加载这个文件，而不需要重启服务器 。

3.4.1.2 web.xml

Web 应用程序描述文件，都是关于是 Web 应用程序的配置文件。所有 Web 应用的 web.xml 文件的父文件。

3.4.1.3 server.xml

是 tomcat 服务器的核心配置文件 ,server.xml 的每一个元素都对应了 tomcat 中的一个组件，通过对 xml 中元素的配置，实现对 tomcat 中的各个组件和端口的配置。

3.4.1.4 tomcat-users.xml

配置访问 Tomcat 的用户以及角色的配置文件。

3.4.2 解决控制台乱码

控制台产生乱码的原因是在 Tomcat 在输出日志中使用的是 UTF-8 编码，而我们中文的 Windows 操作系统使用的是 GBK 编码。由于编码格式不统一，所以出现了乱码。

解决方式：

修改 conf 目录中的 logging.properties 文件重新指定的编码方式。

```
java.util.logging.ConsoleHandler.encoding = GBK
```

3.4.3 修改 Tomcat 监听端口

Tomcat 默认监听端口为 8080。可以通过修改 server.xml 文件来改变 Tomcat 的监听端口。

```
<Connector port="8080" protocol="HTTP/1.1"
                connectionTimeout="20000"
                redirectPort="8443" />
```

3.4.4 配置 Tomcat Manager

3.4.4.1 什么是 Tomcat Manager

Tomcat Manager 是 Tomcat 自带的、用于对 Tomcat 自身以及部署在 Tomcat 上的应用进行管理的 web 应用。默认情况下，Tomcat Manager 是处于禁用状态的。准确的说，Tomcat Manager 需要以用户角色进行登录并授权才能使用相应的功能，不过 Tomcat 并没有配置任何默认的用户，因此我们需要先进行用户配置后才能使用 Tomcat Manager。

3.4.4.2 配置 Tomcat Manager 的访问用户

Tomcat Manager 中没有默认用户，我们需要在 tomcat-users.xml 文件配置。Tomcat Manager 的用户配置需要配置两个部分：角色配置、用户名及密码配置。

3.4.4.2.1 Tomcat Manager 中的角色分类

manager-gui 角色：

允许访问 HTML GUI 和状态页面(即 URL 路径为/manager/html/*)

manager-script 角色：

允许访问文本界面和状态页面(即 URL 路径为/manager/text/*)

manager-jmx 角色：

允许访问 JMX 代理和状态页面(即 URL 路径为/manager/jmxproxy/*)

manager- status 角色：

仅允许访问状态页面(即 URL 路径为/manager/status/*)

3.4.4.2.2 配置用户及角色

```
<role rolename ="admin-gui"/>
<user username ="tomcat" password ="tomcat" role ="admin-gui" />
```

4 Tomcat 版本说明

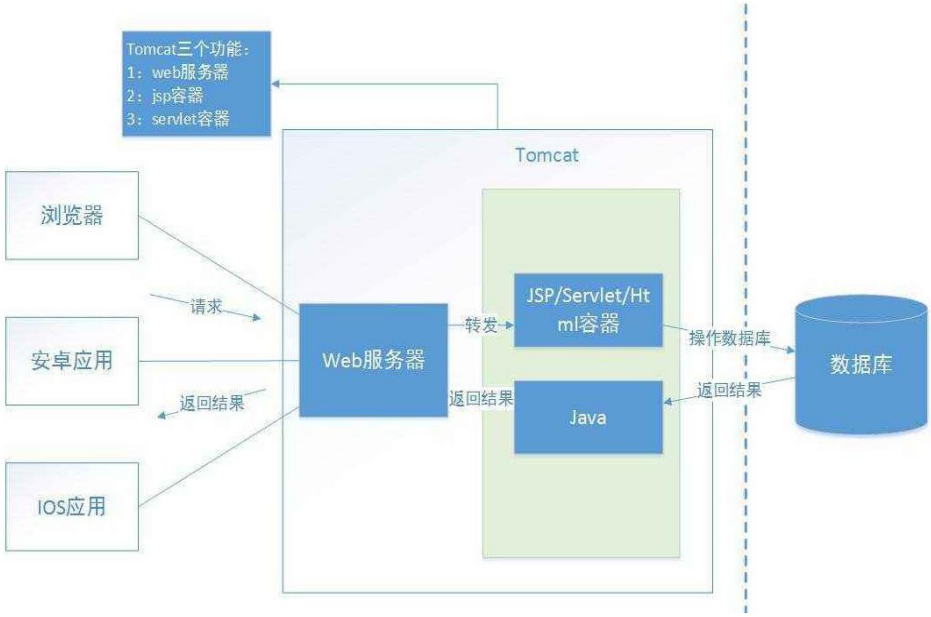
Tomcat 版本	JDK 版本	Servlet 版本	JSP 版本	EL 版本
9.0.x	8 and later	4.0	2.3	3.0
8.5.x	7 and later	3.1	2.3	3.0
8.0.x	7 and later	3.1	2.3	3.0
7.0.x	6 and later	3.0	2.2	2.2
6.0.x	5 and later	2.5	2.1	2.1

5 Tomcat 工作原理

5.1 Tomcat 作用

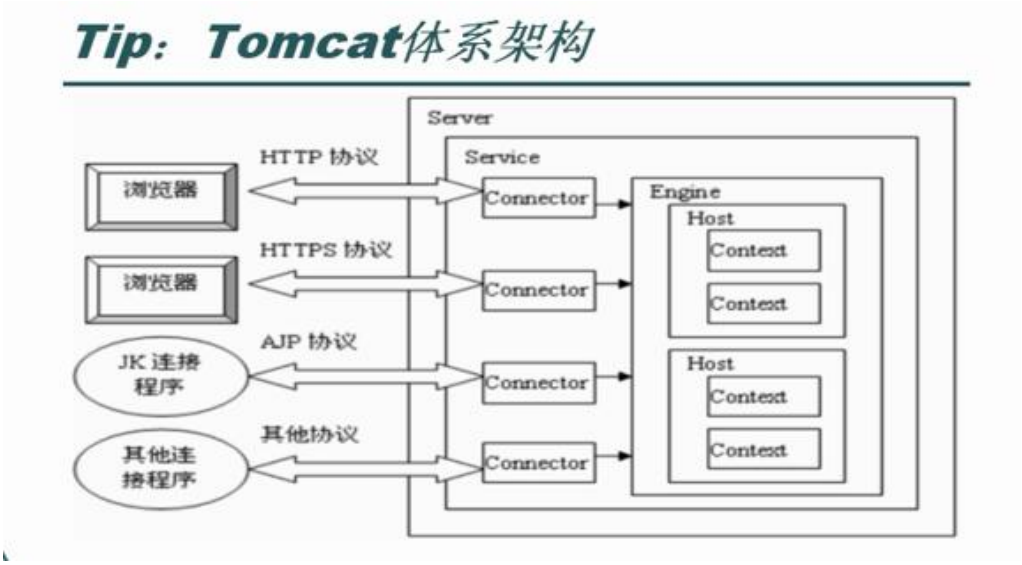
Tomcat 是一个能够处理请求并产生响应的应用程序。Tomcat 实现了 JavaEE 平台下的一些技术规范，所以我们可以在 Tomcat 中运行我们所编写的 Servlet、JSP。

5.2 Tomcat 运行原理



6 Tomcat 架构讲解

6.1 Tomcat 架构图



6.2 Tomcat 组件

6.2.1 Server 组件

启动一个 server 实例(即一个 JVM) ,它监听在 8005 端口以接收 shutdown 命令。Server 的定义不能使用同一个端口，这意味着如果在同一个物理机上启动了多个 Server 实例，必须配置它们使用不同的端口。

```
<Server port="8005" shutdown="SHUTDOWN">
```

port: 接收 shutdown 指令的端口，默认为 8005；

shutdown :发往此 Server 用于实现关闭 tomcat 实例的命令字符串 ,默认为 SHUTDOWN；

6.2.2 Service 组件

Service 主要用于关联一个引擎和与此引擎相关的连接器，每个连接器通过一个特定的端口和协议接收请求并将其转发至关联的引擎进行处理。因此，Service 要包含一个引擎、

一个或多个连接器。

```
<Service name="Catalina">
```

name：此服务的名称，默认为 Catalina；

6.2.3 Connector 组件

支持处理不同请求的组件，一个引擎可以有一个或多个连接器，以适应多种请求方式。默认只开启了处理 Http 协议的连接器。如果需要使用其他协议，需要在 Tomcat 中配置该协议的连接器。

在 Tomcat 中连接器类型通常有 4 种：

1) HTTP 连接器

2) SSL 连接器

3) AJP 1.3 连接器

4) proxy 连接器

```
<Connector port="8888" protocol="HTTP/1.1"
                    connectionTimeout="20000"
                    redirectPort="8443" />
```

port：监听的端口

protocol：连接器使用的协议，默认为 HTTP/1.1;

connectionTimeout：等待客户端发送请求的超时时间，单位为毫秒;

redirectPort：如果某连接器支持的协议是 HTTP，当接收客户端发来的 HTTPS 请求时，则转发至此属性定义的端口;

maxThreads：支持的最大并发连接数，默认为 200 个;

6.2.4 Engine 组件

Engine 是 Servlet 处理器的一个实例，即 servlet 引擎，定义在 server.xml 中的 Service 标

记中。Engine 需要 defaultHost 属性来为其定义一个接收所有发往非明确定义虚拟主机的请求的 host 组件。

```
<Engine name="Catalina" defaultHost="localhost">
```

name：Engine 组件的名称;

defaultHost：Tomcat 支持基于 FQDN(Fully Qualified Domain Name 全限定域名)的虚拟主机，这些虚拟主机可以通过在 Engine 容器中定义多个不同的 Host 组件来实现；但如果此引擎的连接器收到一个发往非非明确定义虚拟主机的请求时则需要将此请求发往一个默认的虚拟主机进行处理，因此，在 Engine 中定义的多个虚拟主机的主机名称中至少要有一个跟 defaultHost 定义的主机名称同名；

6.2.5 Host 组件

位于 Engine 容器中用于接收请求并进行相应处理的虚拟主机。通过该容器可以运行 Servlet 或者 JSP 来处理请求。

```
<Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true">
```

name：虚拟主机的名称，Tomcat 通过在请求 URL 中的域名与 name 中的值匹配，用于查找能够处理该请求的虚拟主机。如果未找到则交给在 Engine 中 defaultHost 指定的主机处理；

appBase：此 Host 的 webapps 目录，即指定存放 web 应用程序的目录的路径；

autoDeploy：在 Tomcat 处于运行状态时放置于 appBase 目录中的应用程序文件是否自动进行 deploy；默认为 true；

unpackWARs：在启用此 webapps 时是否对 WAR 格式的归档文件先进行展开；默认为 true；

6.2.6 Context 组件

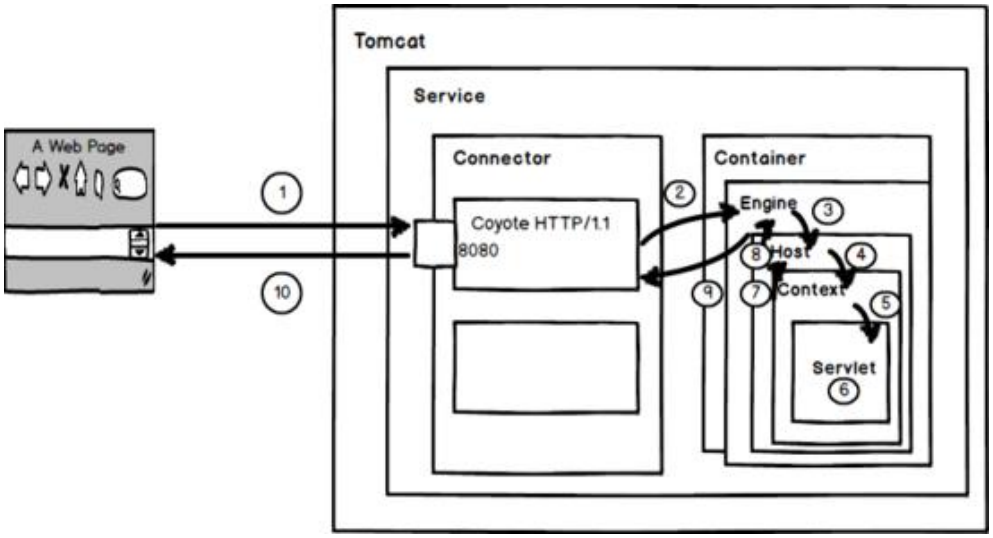
Context 是 Host 的子标签，代表指定一个 Web 应用，它运行在某个指定的虚拟主机（Host）上；每个 Web 应用都是一个 WAR 文件，或文件的目录；

```
<Context path="/test" docBase="D:\bjsxt\itbaizhan.war" />
```

path：context path 既浏览器访问项目的访问路径。

docBase：相应的 Web 应用程序的存放位置，也可以使用相对路径，起始路径为此 Context 所属 Host 中 appBase 定义的路径；

6.3 Tomcat 处理请求过程



1、用户访问 localhost:8080/test/index.jsp，请求被发送到 Tomcat，被监听 8080 端口并处理 HTTP/1.1 协议的 Connector 获得。

2、Connector 把该请求交给它所在的 Service 的 Engine 来处理，并等待 Engine 的回应。

3、Engine 获得请求 localhost/test/index.jsp，匹配所有的虚拟主机 Host。

4、Engine 匹配到名为 localhost 的 Host 虚拟主机来处理/test/index.jsp 请求（即使匹配不到会请求交给默认 Host 处理），Host 会根据/test 匹配它所拥有的所有的 Context。

5、匹配到的 Context 获得请求/index.jsp。

6、构造 HttpServletRequest 对象和 HttpServletResponse 对象，作为参数调用 JspServlet 的 doGet（ ）或 doPost（ ）.执行业务逻辑、数据存储等程序。

7、Context 把执行完之后的结果通过 HttpServletResponse 对象返回给 Host。

8、Host 把 HttpServletResponse 返回给 Engine。

9、Engine 把 HttpServletResponse 对象返回 Connector。

10、Connector 把 HttpServletResponse 对象返回给客户 Browser。

7 在 Tomcat 中配置虚拟主机（Host）

7.1需求

创建 index.html 页面

将 index.html 资源部署到 d 盘的 demo 目录中。

通过 www.itbz.com:8888/itbz/index.html 访问虚拟主机，并访问 index.html

7.2创建页面

创建 index.html 页面

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
 <HEAD>
  <TITLE> ITBZ </TITLE>
  <META NAME="Generator" CONTENT="EditPlus">
  <META NAME="Author" CONTENT="">
  <META NAME="Keywords" CONTENT="">
  <META NAME="Description" CONTENT="">
 </HEAD>
 <BODY>
 Welcome to ITBZ
 </BODY>
</HTML>
```

7.3修改 server.xml 添加 Host 配置

```
<Host name="www.itbz.com" appBase="webapps" unpackWARs="true" autoDeploy="true">
```

7.4添加 Context 配置

```
<Context path="/itbz" docBase="d:/demo"/>
```

7.5修改 windows 的 Host 文件

修改 Windows 系统中的 Host 文件做域名与 IP 的绑定。

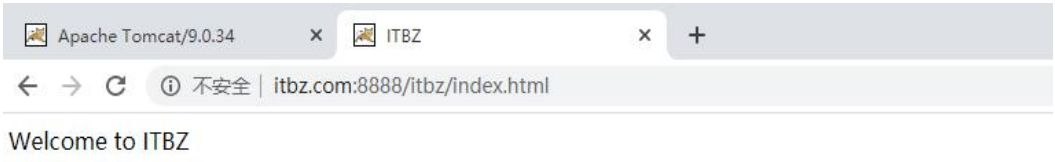
7.5.1 Host 文件位置

C:\Windows\System32\drivers\etc

7.5.2 修改内容

```
127.0.0.1 www.itbz.com
```

7.6访问资源测试结果



四、 Servlet 技术详解

Servlet4.0

1 创建第一个 Servlet 案例

1.1 创建 Servlet

```
package com.bjsxt;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out
                .println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01
Transitional//EN\">");
        out.println("<HTML>");
        out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("  <BODY>");
        out.println("<font color=blue>Hello World</font>");
        out.println("  </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }
}
```

1.2 编译 Servlet

```
D:\>javac -classpath "D:\apache-tomcat-9.0.34\lib\servlet-api.jar" HelloWorld.java
```

1.3 创建 web.xml

1.3.1 什么是 web.xml

项目的部署描述文件，是 JavaWeb 工程的配置文件，通过 web.xml 文件可以配置 servlet、

filter 等技术。Tomcat 启动时会先解析该配置文件获取项目的配置信息。

1.3.2 web.xml 文件中的头信息

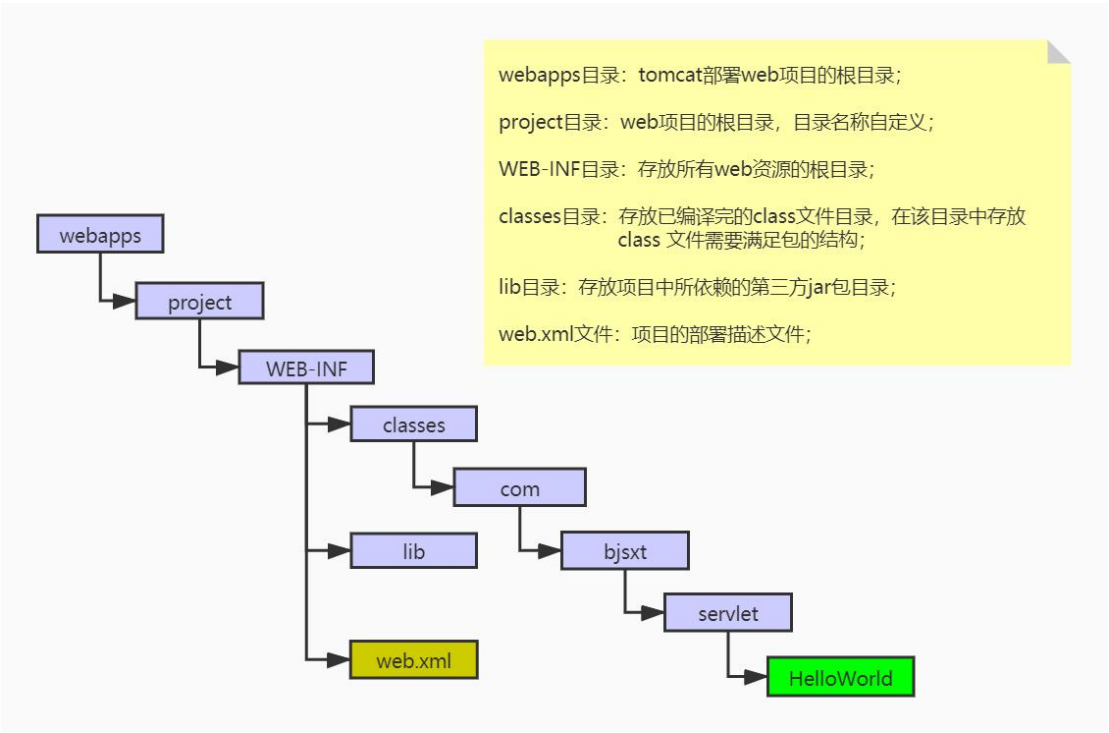
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
        version="4.0">
</web-app>
```

1.3.3 在 web.xml 文件中配置 Servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
        version="4.0">
    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>com.bjsxt.servlet.HelloWorld</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/helloworld.do</url-pattern>
    </servlet-mapping>
</web-app>
```

1.4部署 Servlet

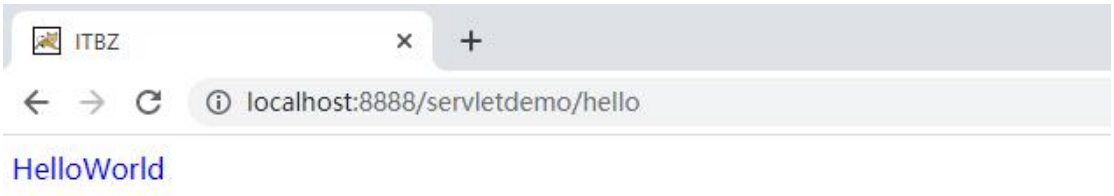
1.4.1 Web 工程目录结构



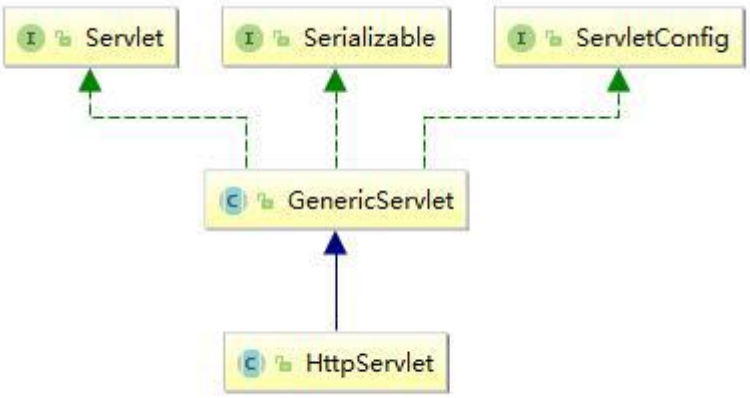
1.5访问 Servlet

http://localhost:8888/servletdemo/hello

1.6访问资源测试结果



2 Servlet 继承结构



2.1 Servlet 接口

1.init()，创建 Servlet 对象后立即调用该方法完成其他初始化工作。

2.service()，处理客户端请求，执行业务操作，利用响应对象响应客户端请求。

3.destroy()，在销毁 Servlet 对象之前调用该方法，释放资源。

4.getServletConfig()，ServletConfig 是容器向 servlet 传递参数的载体。

5.getServletInfo()，获取 servlet 相关信息。

2.2 ServletConfig 接口

Servlet 运行期间，需要一些辅助信息，这些信息可以在 web.xml 文件中，使用一个或多个元素，进行配置。当 Tomcat 初始化一个 Servlet 时，会将该 Servlet 的配置信息，封装到一个 ServletConfig 对象中，通过调用 init(ServletConfig config)方法，将 ServletConfig 对称传递给 Servlet

2.3 GenericServlet 是个抽象类

GenericServlet 是实现了 Servlet 接口的抽象类。在 GenericServlet 中进一步的定义了

Servlet 接口的具体实现，其设计的目的是为了和应用层协议解耦，在 GenericServlet 中包含一个 Service 抽象方法。我们也可以通过继承 GenericServlet 并实现 Service 方法实现请求的处理，但是需要将 ServletReuqest 和 ServletResponse 转为 HttpServletRequest 和 HttpServletResponse。

2.4 HttpServlet 类

继承自 GenericServlet. 针对于处理 HTTP 协议的请求所定制。在 HttpServlet 的 service() 方法中已经把 ServletReuqest 和 ServletResponse 转为 HttpServletRequest 和 HttpServletResponse。 直接使用 HttpServletRequest 和 HttpServletResponse，不再需要强转。实际开发中，直接继承 HttpServlet，并根据请求方式复写 doXxx() 方法即可。

3 Servlet 的生命周期

Servlet 的生命周期是由容器管理的，分别经历三各阶段：

init()：初始化

service()：服务

destroy()：销毁

当客户端浏览器第一次请求 Servlet 时，容器会实例化这个 Servlet，然后调用一次 init 方法，并在新的线程中执行 service 方法处理请求。service 方法执行完毕后容器不会销毁这个 Servlet 而是做缓存处理，当客户端浏览器再次请求这个 Servlet 时，容器会从缓存中直接找到这个 Servlet 对象，并再一次在新的线程中执行 Service 方法。当容器在销毁 Servlet 之前对调用一次 destroy 方法。

4 Servlet 处理请求的原理

当浏览器基于 get 方式请求我们创建 Servlet 时，我们自定义的 Servlet 中的 doGet 方法会被执行。doGet 方法能够被执行并处理 get 请求的原因是，容器在启动时会解析 web 工程中 WEB-INF 目录中的 web.xml 文件，在该文件中我们配置了 Servlet 与 URI 的绑定，容器通过对请求的解析可以获取请求资源的 URI ,然后找到与该 URI 绑定的 Servlet 并做实例化处理(注意：只实例化一次，如果在缓存中能够找到这个 Servlet 就不会再做次实例化处理)。在实例化时会使用 Servlet 接口类型作为引用类型的定义，并调用一次 init 方法，由于 HttpServlet 中重写了该方法所以最终执行的是 HttpServlet 中 init 方法(HttpServlet 中的 Init 方法是一个空的方法体)，然后在新的线程中调用 service 方法。由于在 HttpServlet 中重写了 Service 方法所以最终执行的是 HttpServlet 中的 service 方法。在 service 方法中通过 request.getMethod() 获取到请求方式进行判断如果是 Get 方式请求就执行 doGet 方法，如果是 POST 请求就执行 doPost 方法。如果是基于 GET 方式提交的，并且在我们自定义的 Servlet 中又重写了 HttpServlet 中的 doGet 方法 那么最终会根据 Java 的多态特性转而执行我们自定义的 Servlet 中的 doGet 方法。

5 Servlet 的使用

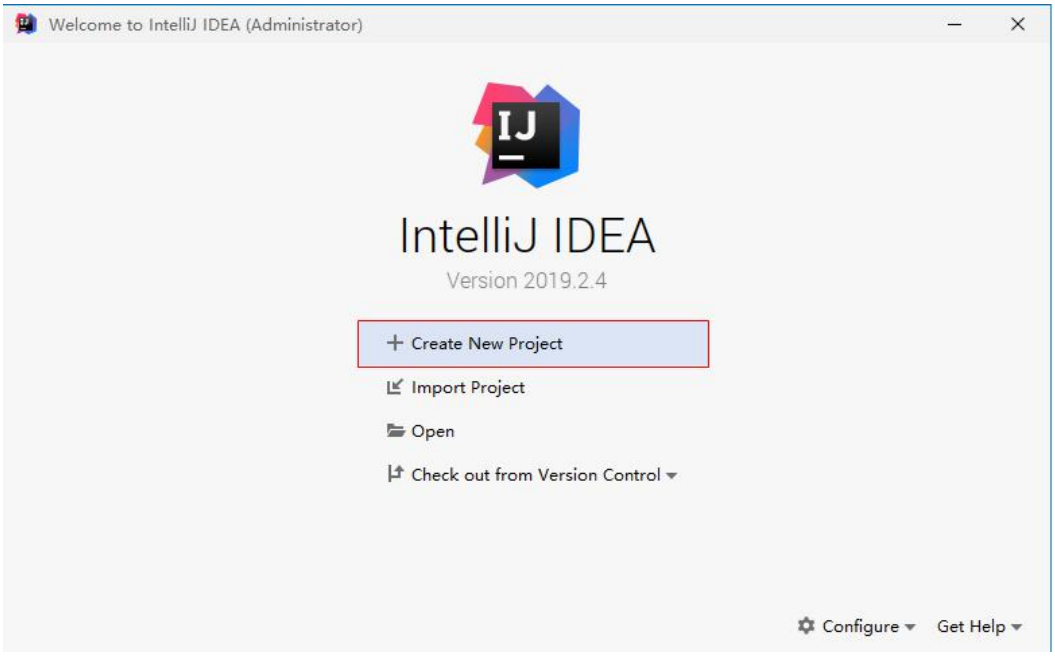
5.1 Servlet 的作用

- 获取用户提交的数据
- 获取浏览器附加的信息
- 处理数据（访问数据库或调用接口）
- 给浏览器产生一个响应
- 在响应中添加附加信息

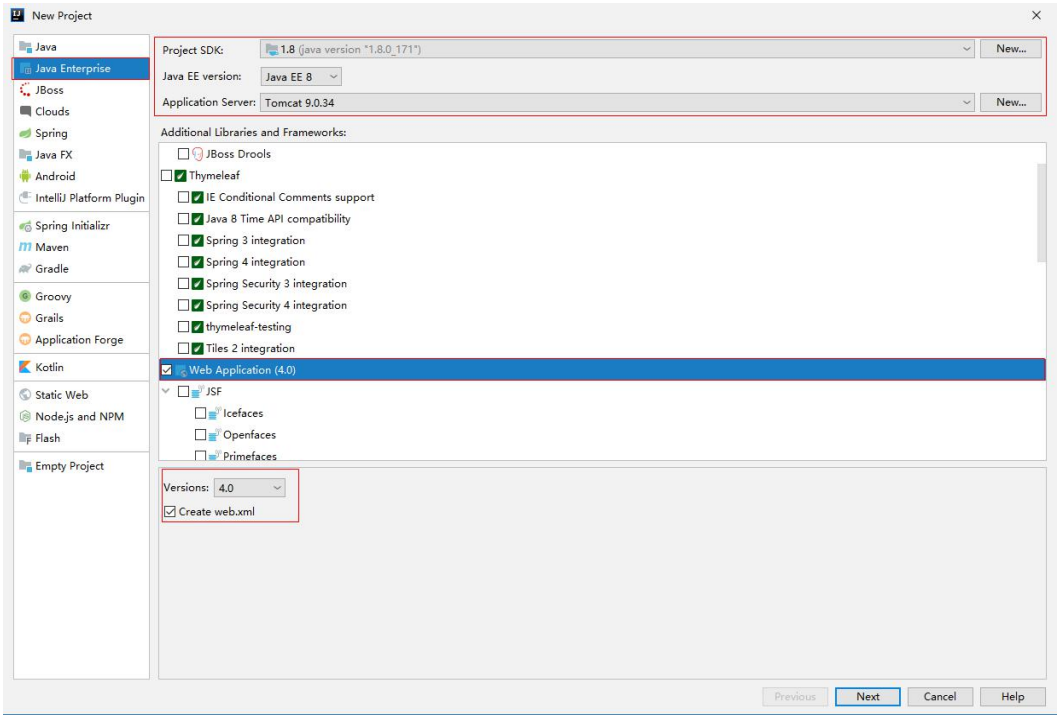
6 在 Idea 中开发 Servlet

6.1在 Idea 创建 Web 工程

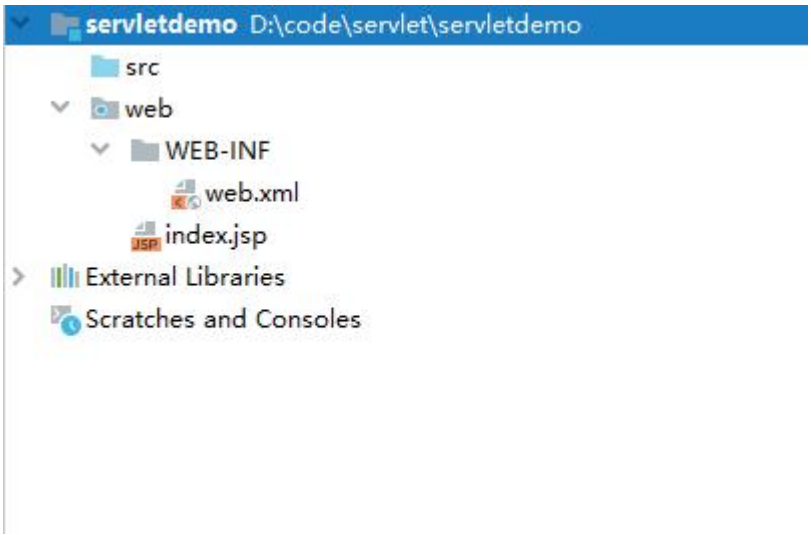
6.1.1 创新项目



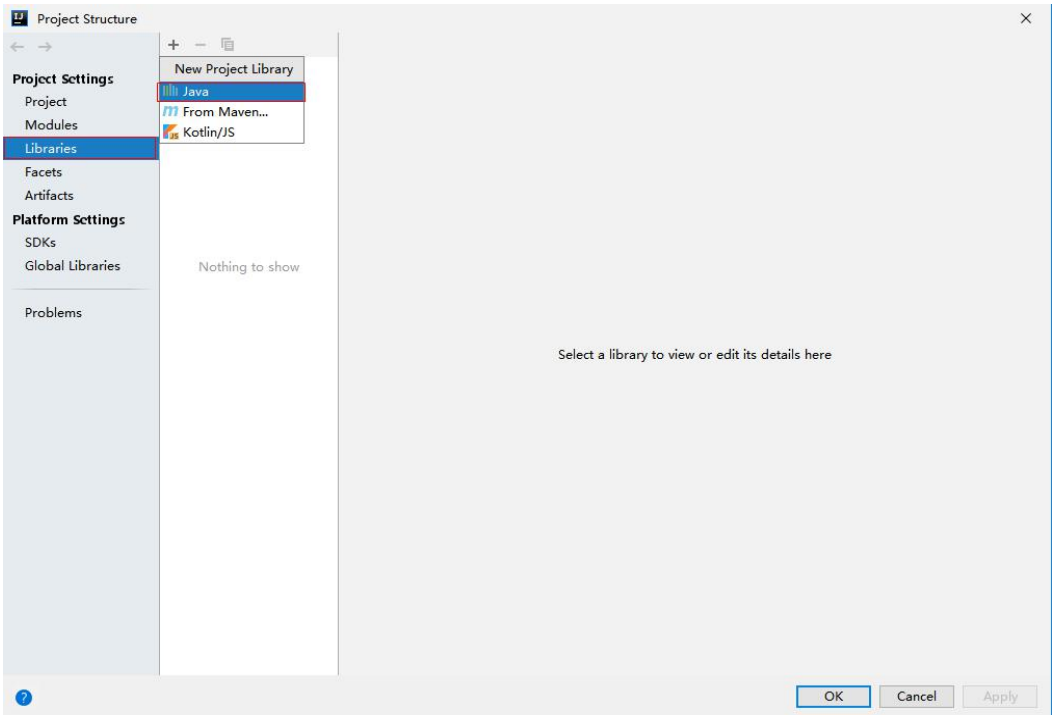
6.1.2 创建 Web 工程

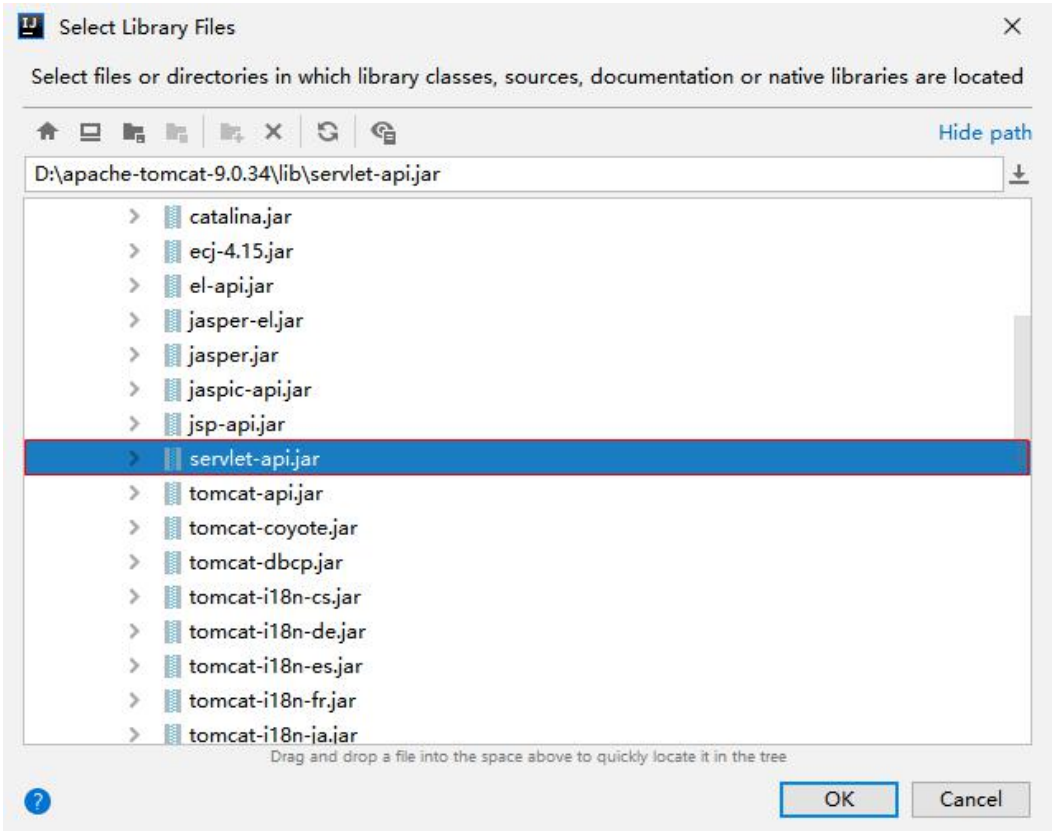


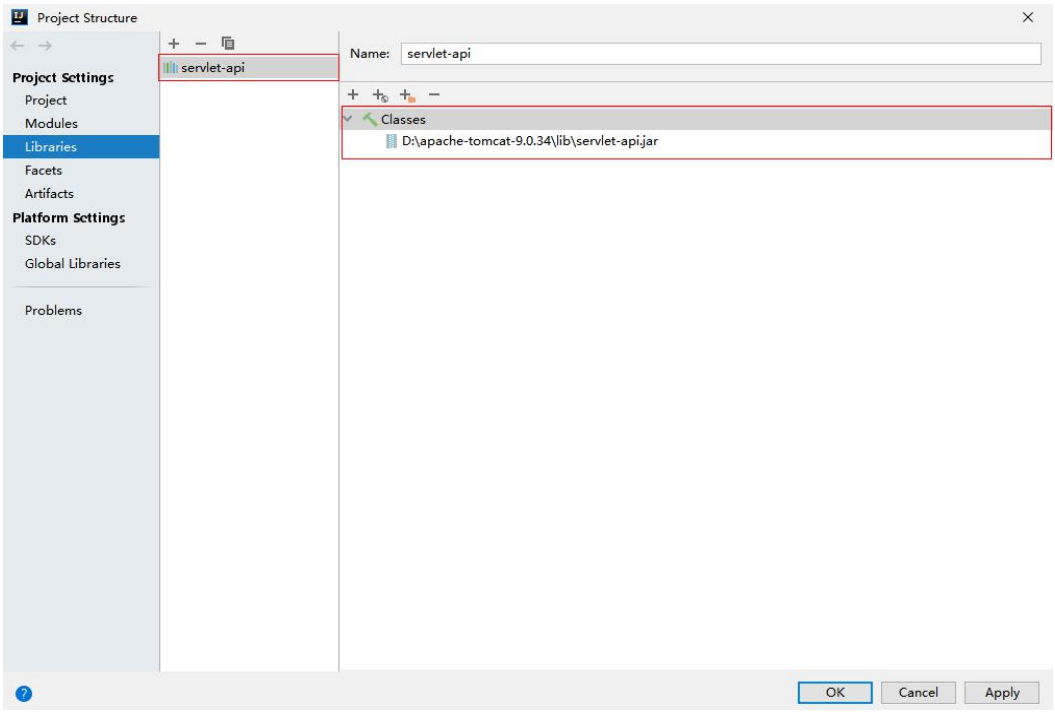
6.1.3 Web 工程结构



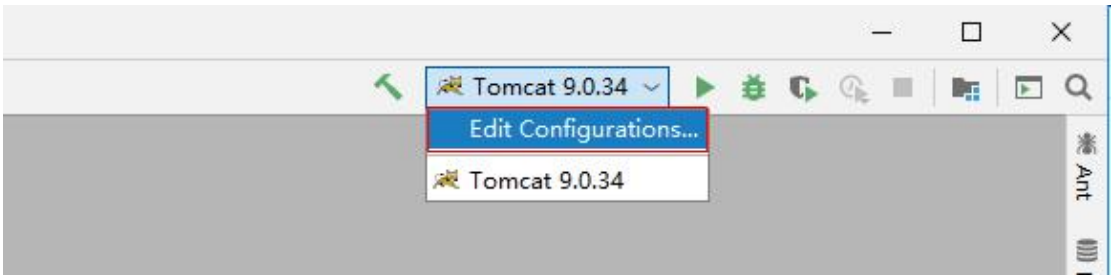
6.1.4 为项目添加 servlet-api.jar

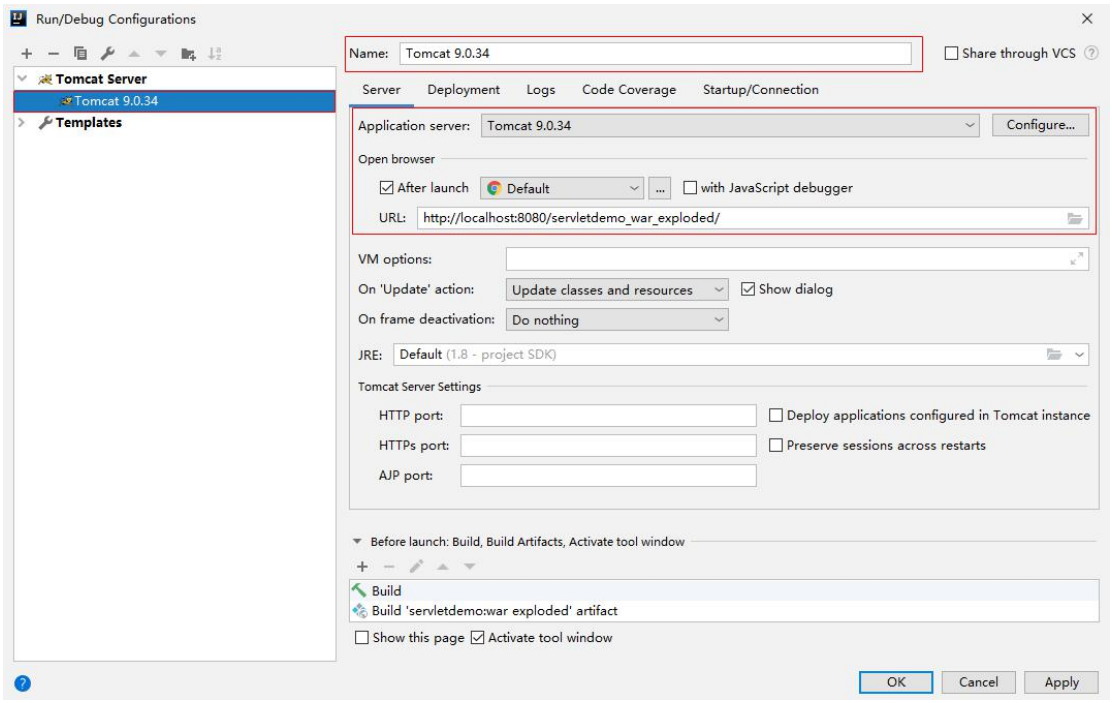




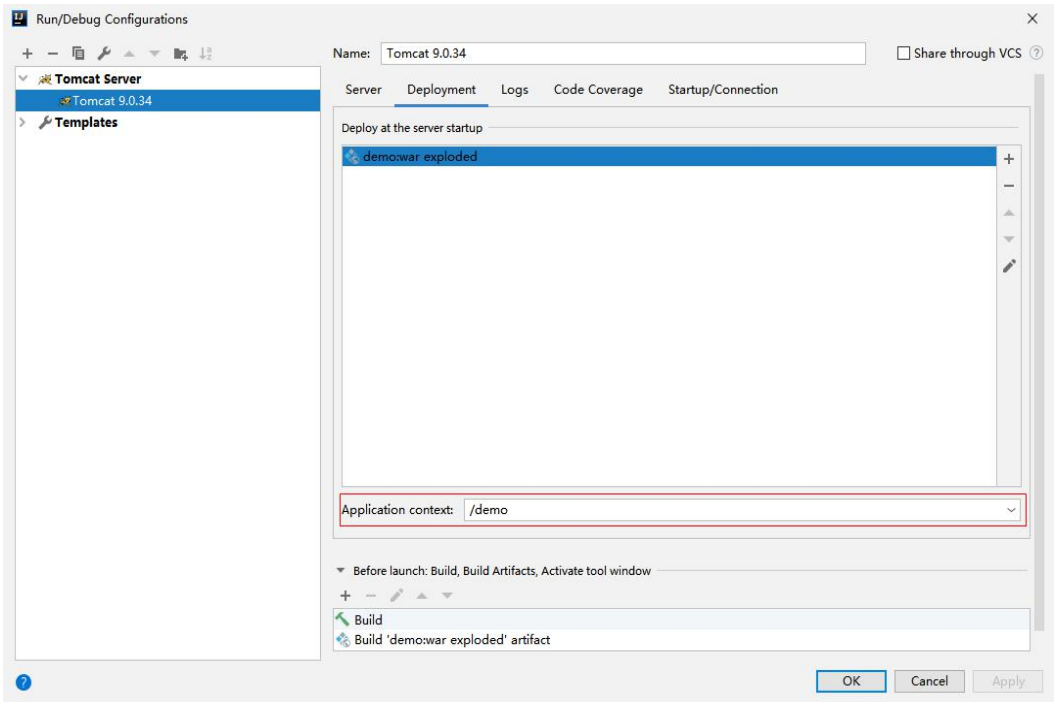


6.1.5 在 Idea 中配置 Tomcat





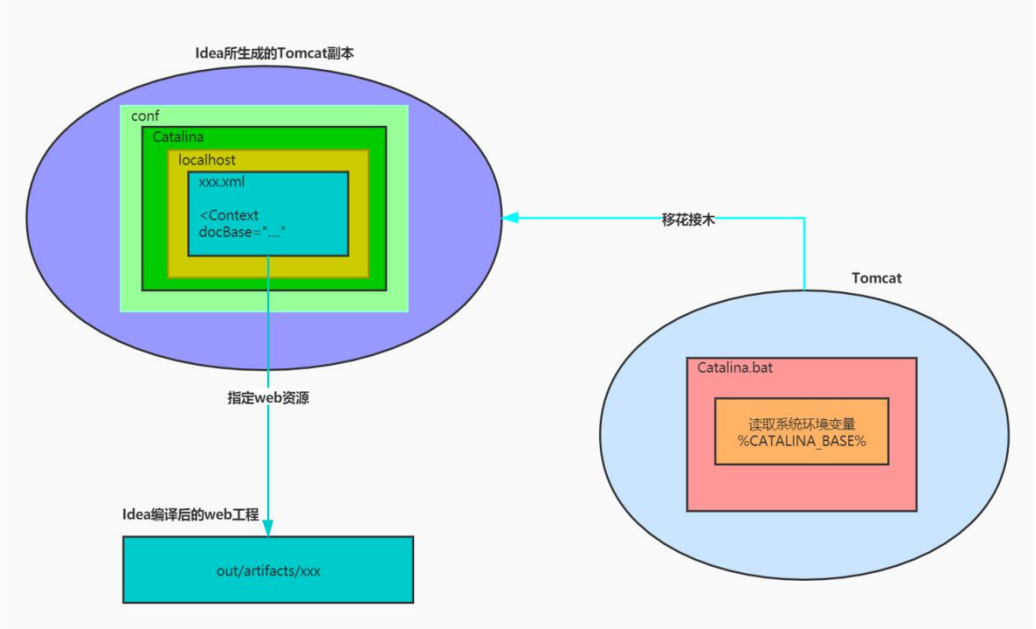
6.1.6 修改项目的访问路径(Content Path)



6.2 Idea 中的 web 项目部署

在 Idea 中默认的并不会把 web 项目真正的部署到 Tomcat 的 webapps 目录中，而是通

过为每个 web 项目创建一个独立的 Tomcat 副本并在 Tomcat 副本中通过的 Tomcat 的 Context 组件完成项目的目录指定 ,在 Context 组件的 docBase 属性中会指定 Idea 对 web 项目编译后的目录 out/artifacts/.....。



6.2.1 默认部署方式

Idea 会在 C:\Users\Administrator\.IntelliJIdea2019.2\system\tomcat 中为每个 Web 项目创建一个独立的 Tomcat 副本。

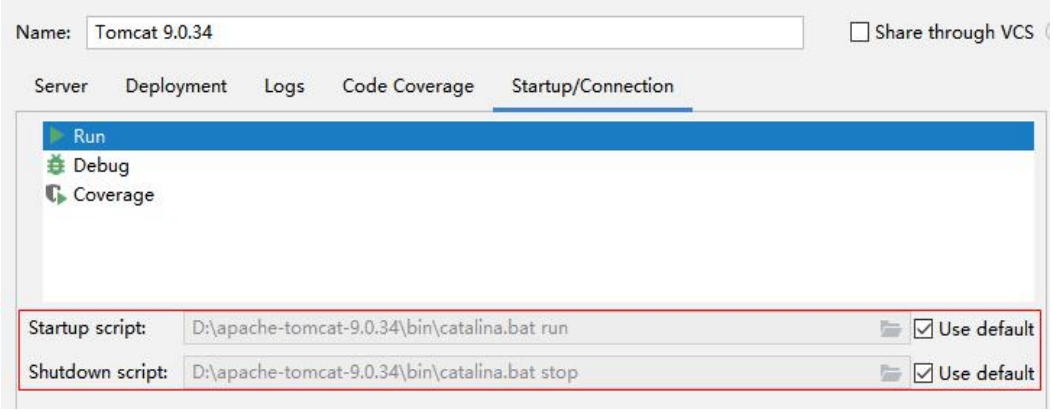
C:\Users\Administrator\.IntelliJIdea2019.2\system\tomcat\Tomcat_9_0_34_demo_4\conf\C

atalina\localhost 目录中生成一个该项目的 xml 文件名称为：”项目名.xml”。

在 xml 文件中指定 web 项目编译完后的 artifacts 目录的位置。

```
<Context path="/demo"
docBase="D:\code\servlet\demo\out\artifacts\demo_war_exploded" />
```

Idea 通过执行 Tomcat 的 catalina.bat 启动脚本启动 Tomcat，通过启动参数来指定启动 Tomcat 副本运行指定目录中的 web 项目。



Idaa 在启动 Tomcat 之前会先在操作系统中设置一些临时环境变量，这些变量会被 Tomcat 的启动脚本所读取。

```
Using CATALINA_BASE:   "C:\Users\Administrator\.IntelliJIdea2019.2\system\tomcat\T
Using CATALINA_HOME:   "D:\apache-tomcat-9.0.34"
Using CATALINA_TMPDIR: "D:\apache-tomcat-9.0.34\temp"
Using JRE_HOME:        "C:\Program Files\Java\jdk1.8.0_171"
Using CLASSPATH:       "D:\apache-tomcat-9.0.34\bin\bootstrap.jar;D:\apache-tomcat
```

CATALINA_BASE：是 Tomcat 副本的工作目录

CATALINA_HOME：是 Tomcat 的安装目录

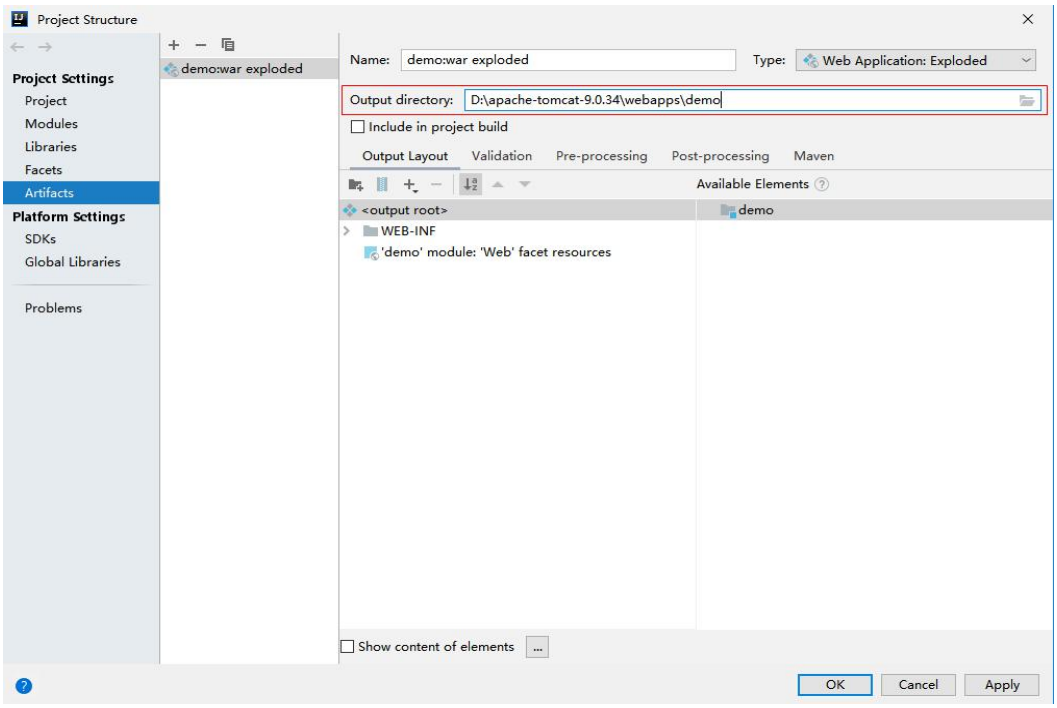
在 Catalina.bat 启动脚本运行时，会先去判断脚本中的 CATALINA_HOME 以及 CATALINA_BASE 是否有默认值，如果没有则直接读取系统环境变量中的值作为他们的默认值。由于 Idea 在启动 Tomcat 之前已经设置了临时环境变量，所以 tomcat 在启动后就会运行部署在 Tomcat 副本中的 web 项目。

```
134 set "CURRENT_DIR=%cd%"
135 if not "%CATALINA_HOME%" == "" goto gotHome
136 set "CATALINA_HOME=%CURRENT_DIR%"
137 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
138 cd ..
139 set "CATALINA_HOME=%cd%"
140 cd "%CURRENT_DIR%"
141 :gotHome
142
143 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
144 echo The CATALINA_HOME environment variable is not defined correctly
145 echo This environment variable is needed to run this program
146 goto end
147 :okHome
148
149 rem Copy CATALINA_BASE from CATALINA_HOME if not defined
150 if not "%CATALINA_BASE%" == "" goto gotBase
151 set "CATALINA_BASE=%CATALINA_HOME%"
152 :gotBase
153
```

6.2.2 将 web 项目部署到 Tomcat 的 webapps 中

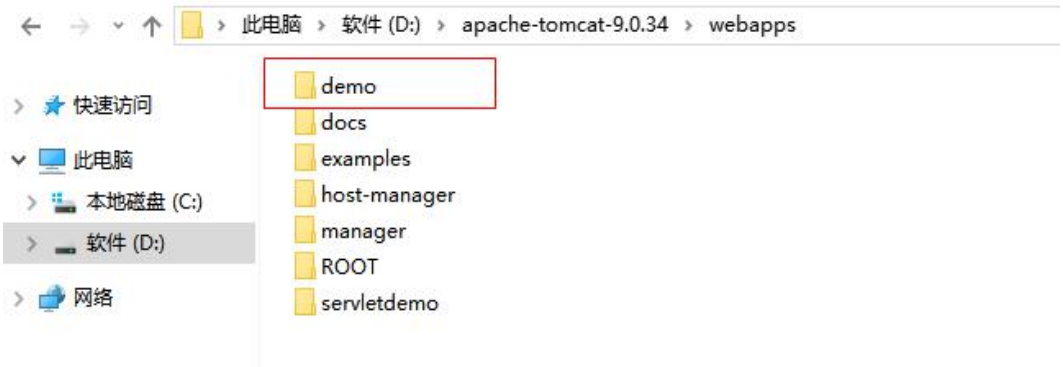
点击项目结构选项

指定输出 artifacts 的目录为 Tomcat 的 webapps 中的 demo 目录。



启动 Tomcat，查看 demo 目录中的内容。

在 tomcat 的 webapps 中创建一个目录。



7 HttpServletRequest 对象

HttpServletRequest 对象代表客户端浏览器的请求，当客户端浏览器通过 HTTP 协议访问

服务器时，HTTP 请求中的所有信息都会被 Tomcat 所解析并封装在这个对象中，通过这个对象提供的方法，可以获得客户端请求的所有信息。

7.1 获取请求信息

req.getRequestURL()

返回客户端浏览器发出请求时的完整 URL。

req.getRequestURI()

返回请求行中指定资源部分。

req.getRemoteAddr()

返回发出请求的客户机的 IP 地址。

req.getLocalAddr()

返回 WEB 服务器的 IP 地址。

req.getLocalPort()

返回 WEB 服务器处理 Http 协议的连接器所监听的端口。

7.2 获取请求数据

7.2.1 在 Servlet 获取请求数据的方式

req.getParameter("key")

根据 key 获取指定 value。

```
String str = req.getParameter("key");
```

7.2.2 获取复选框(checkbox 组件)中的值

req.getParameterValues("checkboxkey")

获取复选框(checkbox 组件)中的值，返回一个 String[]。

```
String[] userlikes = req.getParameterValues("checkboxkey");
```

7.2.3 获取所有提交数据的 key

req.getParameterNames()

获取请求中所有数据的 key，该方法返回一个枚举类型。

```
Enumeration<String> parameterNames = req.getParameterNames();
```

7.2.4 使用 Map 结构获取提交数据

req.getParameterMap()

获取请求中所有的数据并存放到一个 Map 结构中，该方法返回一个 Map，其中 key 为 String 类型 value 为 String[]类型。

```
Map<String, String[]> parameterMap = req.getParameterMap();
```

7.3 设置请求编码

req.setCharacterEncoding("utf-8")

请求的数据包基于字节在网络上传输，Tomcat 接收到请求的数据包后会将数据包中的字节转换为字符。在 Tomcat 中使用的是 ISO-8859-1 的单字节编码完成字节与字符的转换，所以数据中含有中文就会出现乱码，可以通过 req.setCharacterEncoding("utf-8")方法来对提交的数据根据指定的编码方式重新做编码处理。

7.4 资源访问方式

7.4.1 绝对路径

绝对路径访问资源表示直接以”/”作为项目的 Content Path。该方式适用于以”/”作为项目的 Content Path。

```
<form action="/getInfo.do" method="post">
```

7.4.2 相对路径

相对路径访问资源表示会相对于项目的 Content Path 作为相对路径。该方式适用于为项目指定的具体的 Content Path。

```
<form action="getInfo.do" method="post">
```

7.5 获取请求头信息

7.5.1 获取请求头信息

req.getHeader("headerKey")

根据请求头中的 key 获取对应的 value。

```
String headerValue = req.getHeader("headerKey");
```

req.getHeaderNames()

获取请求头中所有的 key，该方法返回枚举类型。

```
Enumeration<String> headerNames = req.getHeaderNames();
```

7.5.2 获取请求头案例

需求：编写一个 Servlet，如果浏览器的语言是 zh-CN，显示“你好，聪明的中国人！”，如果浏览器的语言设置为 en-US,那么则显示“Hello，American”。

7.6 HttpServletRequest 对象的生命周期

当有请求到达 Tomcat 时，Tomcat 会创建 HttpServletRequest 对象，并将该对象通过参数的方式传递到我们 Servlet 的方法中，当处理请求处理完毕并产生响应后该对象生命周期结束。

8 HttpServletResponse 对象

HttpServletResponse 对象代表服务器的响应。这个对象中封装了响应客户端浏览器的流

对象，以及向客户端浏览器响应的响应头、响应数据、响应状态码等信息。

8.1设置响应类型

resp.setContentType("MIME")

该方法可通过 MIME-Type 设置响应类型。

Type	Meaning
application/msword	Microsoft Word document
application/octet-stream	Unrecognized or binary data
application/pdf	Acrobat (.pdf) file
application/postscript	PostScript file
application/vnd.lotus-notes	Lotus Notes file
application/vnd.ms-excel	Excel spreadsheet
application/vnd.ms-powerpoint	PowerPoint presentation
application/x-gzip	Gzip archive
application/x-java-archive	JAR file
application/x-java-serialized-object	Serialized Java object
application/x-java-vm	Java bytecode (.class) file
application/zip	Zip archive
application/json	JSON
audio/basic	Sound file in .au or .snd format
audio/midi	MIDI sound file
audio/x-aiff	AIFF sound file
audio/x-wav	Microsoft Windows sound file
image/gif	GIF image
image/jpeg	JPEG image
image/png	PNG image
image/tiff	TIFF image
image/x-xbitmap	X Windows bitmap image
text/css	HTML cascading style sheet

Type	Meaning
text/html	HTML document
text/plain	Plain text
text/xml	XML
video/mpeg	MPEG video clip
video/quicktime	QuickTime video clip

8.1.1 设置字符型响应

常见的字符型响应类型：

resp.setContentType("text/html")

设置响应类型为文本型，内容含有 html 字符串，是默认的响应类型

resp.setContentType("text/plain")

设置响应类型为文本型，内容是普通文本。

resp.setContentType("application/json")

设置响应类型为 JSON 格式的字符串。

8.1.2 设置字节型响应

常见的字节型响应：

resp.setContentType("image/jpeg")

设置响应类型为图片类型，图片类型为 jpeg 或 jpg 格式。

resp.setContentType("image/gif")

设置响应类型为图片类型，图片类型为 gif 格式。

8.2设置响应编码

response.setCharacterEncoding("utf-8");

设置服务端为浏览器产生响应的响应编码，服务端会根据此编码将响应内容的字符转换

为字节。

response.setContentType("text/html;charset=utf-8");

设置服务端为浏览器产生响应的响应编码，服务端会根据此编码将响应内容的字符转换为字节。同时客户端浏览器会根据此编码方式显示响应内容。

8.3 在响应中添加附加信息

8.3.1 重定向响应

response.sendRedirect(URL 地址)

重定向响应会在响应头中添加一个 Location 的 key 对应的 value 是给定的 URL。客户端浏览器在解析响应头后自动向 Location 中的 URL 发送请求。

重定向响应特点：

重定向会产生两次请求两次响应。

重定向的 URL 是有客户端浏览器发送的。

8.3.2 重定向响应案例

需求：创建一个搜索页面，通过百度搜索引擎完成内容搜索。

8.3.3 文件下载

在实现文件下载时，我们需要修改响应头，添加附加信息。

```
response.setHeader("Content-Disposition", "attachment; filename="+文件名);
```

Content-Disposition:attachment

该附加信息表示作为对下载文件的一个标识字段。不会在浏览器中显示而是直接做下载处理。

filename=文件名

表示指定下载文件的文件名。

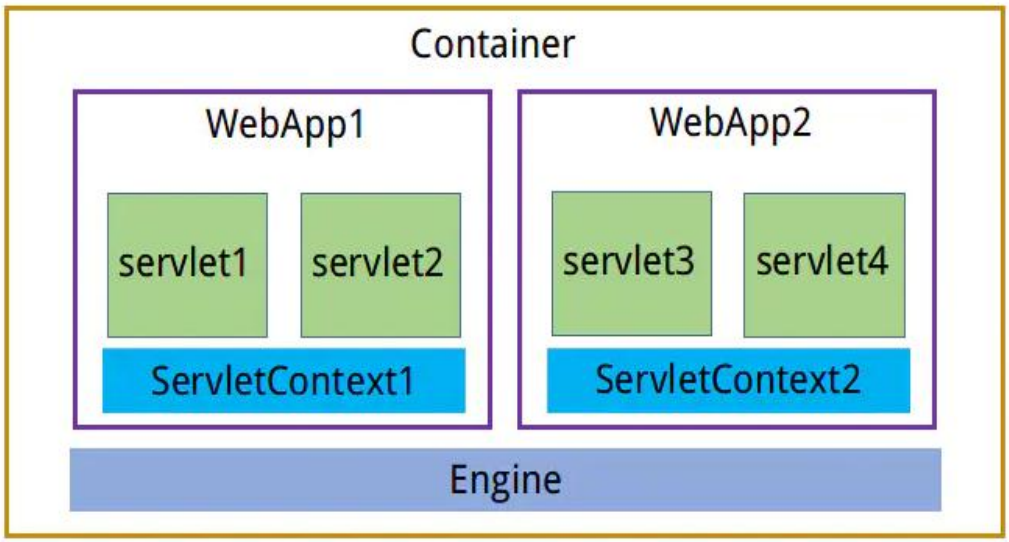
8.3.4 解决文件名中文乱码问题

```
resp.addHeader("Content-Disposition","attachment;filename="+new
String(file.getName().getBytes("gbk"),"iso-8859-1"));
```

9 ServletContext 对象

9.1 ServletContext 对象介绍

ServletContext 官方叫 Servlet 上下文。服务器会为每一个 Web 应用创建一个 ServletContext 对象。这个对象全局唯一，而且 Web 应用中的所有 Servlet 都共享这个对象。所以叫全局应用程序共享对象。



9.2 ServletContext 对象的作用

- 相对路径转绝对路径
- 获取容器的附加信息

- 读取配置信息
- 全局容器

9.3 ServletContext 对象的使用

9.3.1 相对路径转绝对路径

context.getRealPath("path")

该方法可以将一个相对路径转换为绝对路径 ,在文件上传与下载时需要用到该方法做路径的转换。

9.3.2 获取容器的附加信息

servletContext.getServerInfo()

返回 Servlet 容器的名称和版本号

servletContext.getMajorVersion()

返回 Servlet 容器所支持 Servlet 的主版本号。

servletContext.getMinorVersion()

返回 Servlet 容器所支持 Servlet 的副版本号。

9.3.3 获取 web.xml 文件中的信息

```
<context-param>
    <param-name>key</param-name>
    <param-value>value</param-value>
</context-param>
```

servletContext.getInitParameter("key")

该方法可以读取 web.xml 文件中<context-param>标签中的配置信息。

servletContext.getInitParameterNames()

该方法可以读取 web.xml 文件中所有 param-name 标签中的值。

9.3.4 全局容器

servletContext.setAttribute("key",ObjectValue)

向全局容器中存放数据。

servletContext.getAttribute("key")

从全局容器中获取数据。

servletContext.removeAttribute("key")

根据 key 删除全局容器中的 value。

9.4 ServletContext 对象生命周期

当容器启动时会创建 ServletContext 对象并一直缓存该对象，直到容器关闭后该对象生命周期结束。ServletContext 对象的生命周期非常长，所以在使用全局容器时不建议存放业务数据。

10 ServletConfig 对象

ServletConfig 对象对应 web.xml 文件中的<servlet>节点。当 Tomcat 初始化一个 Servlet 时，会将该 Servlet 的配置信息，封装到一个 ServletConfig 对象中。我们可以通过该对象读取<servlet>节点中的配置信息

```
<servlet>
    <servlet-name>servletName</servlet-name>
    <servlet-class>servletClass</servlet-class>
    <init-param>
        <param-name>key</param-name>
        <param-value>value</param-value>
    </init-param>
</servlet>
```

servletConfig.getInitParameter("key")

该方法可以读取 web.xml 文件中<servlet>标签中<init-param>标签中的配置信息。

servletConfig.getInitParameterNames()

该方法可以读取 web.xml 文件中当前<servlet>标签中所有<init-param>标签中的值。

11 Cookie 对象与 HttpSession 对象

Cookie 对象与 HttpSession 对象的作用是维护客户端浏览器与服务端的会话状态的两个对象。由于 HTTP 协议是一个无状态的协议，所以服务端并不会记录当前客户端浏览器的访问状态，但是在有些时候我们是需要服务端能够记录客户端浏览器的访问状态的，如获取当前客户端浏览器的访问服务端的次数时就需要会话状态的维持。在 Servlet 中提供了 Cookie 对象与 HttpSession 对象用于维护客户端与服务端的会话状态的维持。二者不同的是 Cookie 是通过客户端浏览器实现会话的维持，而 HttpSession 是通过服务端来实现会话状态的维持。

11.1Cookie 对象

11.1.1 Cookie 对象的特点

- Cookie 使用字符串存储数据
- Cookie 使用 Key 与 Value 结构存储数据
- 单个 Cookie 存储数据大小限制在 4097 个字节
- Cookie 存储的数据中不支持中文，Servlet4.0 中支持
- Cookie 是与域名绑定所以不支持跨一级域名访问
- Cookie 对象保存在客户端浏览器或系统磁盘中
- Cookie 分为持久化 Cooke 与状态 Cookie
- 浏览器在保存同一域名所返回 Cookie 的数量是有限的。不同浏览器支持的数量不同，

Chrome 浏览器为 50 个

- 浏览器每次请求时都会把与当前访问的域名相关的 Cookie 在请求中提交到服务端。

11.1.2 Cookie 对象的使用

11.1.2.1 Cookie 对象的创建

Cookie cookie = new Cookie("key","value")

通过 new 关键字创建 Cookie 对象

response.addCookie(cookie)

通过 HttpServletResponse 对象将 Cookie 写回给客户端浏览器。

11.1.2.2 获取 Cookie 中的数据

通过 HttpServletRequest 对象获取 Cookie，返回 Cookie 数组。

Cookie[] cookies = request.getCookies()

11.1.2.3 解决 Cookie 不支持中文

在 Servlet4.0 版本之前的 Cookie 中是不支持中文存储的，如果存储的数据中含有中文，代码会直接出现异常。我们可以通过对含有中文的数据重新进行编码来解决该问题。在 Servlet4.0 中的 Cookie 是支持中文存储的。

```
java.lang.IllegalArgumentException: Control character in cookie value or attribute.
```

URLEncoder.encode("content","code")

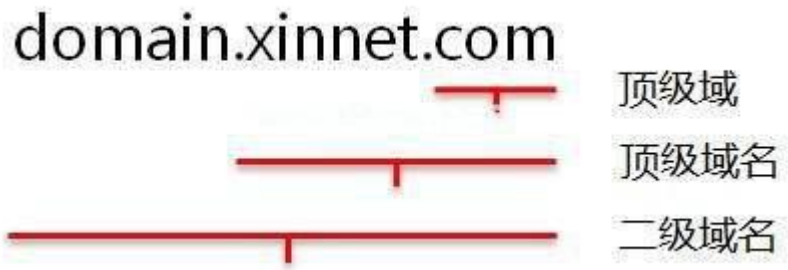
将内容按照指定的编码方式做 URL 编码处理。

URLDecoder.decode("content","code")

将内容按照指定的编码方式做 URL 解码处理。

11.1.2.4 Cookie 跨域问题

域名分类：域名分为顶级域、顶级域名(一级域名)、二级域名。



域名等级的区别：一级域名比二级域名更高级，二级域名是依附于一级域名之下的附属分区域名，即二级域名是一级域名的细化分级。例如：baidu.com 为一级域名，news.baidu.com 为二级域名。

Cookie 不支持一级域名的跨域，支持二级域名的跨域。

11.1.2.5 状态 Cookie 与持久化 Cookie

状态 Cookie：浏览器会缓存 Cookie 对象。浏览器关闭后 Cookie 对象销毁。

持久化 Cookie：浏览器会对 Cookie 做持久化处理，基于文件形式保存在系统的指定目录中。在 Windows10 系统中为了安全问题不会显示 Cookie 中的内容。

当 Cookie 对象创建后默认为状态 Cookie。可以使用 Cookie 对象下的 cookie.setMaxAge(60) 方法设置失效时间，单位为秒。一旦设置了失效时间，那么该 Cookie 为持久化 Cookie，浏览器会将 Cookie 对象持久化到磁盘中。当失效时间到达后文件删除。

11.1.3 通过 Cookie 实现客户端与服务端会话的维持

需求：当客户端浏览器第一次访问 Servlet 时返回“您好，欢迎您第一次访问！”，第二次访问时返回“欢迎您回来！”。

11.1.4 Cookie 总结

Cookie 对于存储内容是基于明文的方式存储的，所以安全性很低。不要在 Cookie 中存放敏感数据。在数据存储时，虽然在 Servlet4.0 中 Cookie 支持中文，但是建议对 Cookie 中存放的内容做编码处理，也可提高安全性。

11.2HttpSession 对象

11.2.1 HttpSession 对象的特点

HttpSession 保存在服务端

HttpSession 可以存储任何类型的数据

HttpSession 使用 Key 与 Value 结构存储数据

HttpSession 存储数据大小无限制

11.2.2 HttpSession 对象的使用

11.2.2.1 HttpSession 对象的创建

HttpSession 对象的创建是通过 request.getSession()方法来创建的。客户端浏览器在请求服务端资源时，如果在请求中没有 SessionID，getSession()方法将会为这个客户端浏览器创建一个新的 HttpSession 对象，并为这个 HttpSession 对象生成一个 SessionID，在响应中通过 Cookie 写回给客户端浏览器，如果在请求中包含了 SessionID，getSession()方法则根据这个 ID 返回与这个客户端浏览器对应的 HttpSession 对象。

getSession()方法还有一个重载方法 getSession(true|false)。当参数为 true 时与 getSession()方法作用相同。当参数为 false 时则只去根据 SessionID 查找是否有与这个客户端浏览器对应

的 HttpSession，如果有则返回，如果没有 SessionID 则不会创建新的 HttpSession 对象。

11.2.2.2 获取 HttpSession 中的数据

session.setAttribute("key",value)

将数据存储到 HttpSession 对象中

Object value = session.getAttribute("key")

根据 key 获取 HttpSession 中的数据，返回 Object

Enumeration<String> attributeNames = session.getAttributeNames()

获取 HttpSession 中所有的 key，返回枚举类型

session.removeAttribute("key")

根据 key 删除 HttpSession 中的数据

String id = session.getId()

根据获取当前 HttpSession 的 SessionID，返回字符串类型

11.2.2.3 HttpSession 的销毁方式

HttpSession 的销毁方式有两种：

- 通过 web.xml 文件指定超时时间
- 通过 HttpSession 对象中的 invalidate()方法销毁当前 HttpSession 对象

我们可以在 web.xml 文件中指定 HttpSession 的超时时间，当到达指定的超时时间后，容器就会销该 HttpSession 对象，单位为分钟。该时间对整个 web 项目中的所有 HttpSession 对象有效。时间的计算方式是根据最后一次请求时间作为起始时间。如果有哪个客户端浏览器对应的 HttpSession 的失效时间已到，那么与该客户端浏览器对应的 HttpSession 对象就会被销毁。其他客户端浏览器对应的 HttpSession 对象会继续保存不会被销毁。

```
<session-config>
    <session-timeout>1</session-timeout>
</session-config>
```

我们也可以在 Tomcat 的 web.xml 文件中配置 HttpSession 的销毁时间。如果在 Tomcat 的 web.xml 文件中配置了 HttpSession 的超时时间对应的是 Tomcat 中所有的 Web 项目都有效。相当于配置了全局的 HttpSession 超时时间。如果我们在 Web 项目中配置了超时时间，那么会以 Web 项目中的超时时间为准。

```
633     <session-config>
634         <session-timeout>30</session-timeout>
635     </session-config>
636
```

invalidate()方法是 HttpSession 对象中所提供的用于销毁当前 HttpSession 的方法。我们通过调用该方法可以销毁当前 HttpSession 对象。

11.2.3 通过 HttpSession 实现客户端与服务端会话的维持

需求：当客户端浏览器第一次访问 Servlet 时返回“您好，欢迎您第一次访问！”，第二次访问时返回“欢迎您回来！”。

11.2.4 HttpSession 生命周期

在 HttpSession 对象生命周期中没有固定的创建时间与销毁时间。何时创建取决于我们什么时候第一次调用了 getSession()或 getSession(true)的方法。HttpSession 对象的销毁时间取决于超时时间的到达以及调用了 invalidate()方法。如果没有超时或者没有调用 invalidate()方法，那么 HttpSession 会一直存储。默认超时时间为 30 分钟(Tomcat 的 web.xml 文件配置的时间就是默认超时时间)。

11.2.5 HttpSession 对象总结

11.2.5.1 HttpSession 与 Cookie 的区别：

- cookie 数据存放在客户的浏览器或系统的文件中，而 HttpSession 中的数据存放在服务器中。
- cookie 不安全，而 HttSession 是安全的。
- 单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个域名保存 cookie 的数量。而 HttpSession 没有容量以及数量的限制。

11.2.5.2 HttpSession 的使用建议

HttpSession 对象是保存在服务端的，所以安全性较高。我们可以在 HttpSession 对象中存储数据，但是由于 HttpSession 对象的生命周期不固定，所以不建议存放业务数据。一般情况下我们只是存放用户登录信息。

12 自启动 Servlet

12.1自启动 Servlet 特点

自动启动 Servlet 表示在 Tomcat 启动时就会实例化这个 Servlet ,他的实例化过程不依赖于请求，而是依赖容器的启动。

可以通过在 web.xml 中的<servlet>标签中通过<load-on-startup>1</load-on-startup>配置自启动 Servlet。

12.2通过自启动 Servlet 实现配置信息的读取

需求：修改文件下载案例，通过自启动 Servlet 读取配置信息。

13 Servlet 线程安全问题

在 Servlet 中使用的是多线程方式来执行 service()方法处理请求，所以我们在使用 Servlet 时需要考虑到线程安全问题，在多线程中对于对象中的成员变量是最不安全的，所以不要在 Servlet 中通过成员变量的方式来存放数据，如果一定要使用成员变量存储数据，在对数据进行操作时需要使用线程同步的方式来解决线程安全问题，避免出现数据张冠李戴现象。

14 Servlet 的 url-pattern 配置

14.1URL 的匹配规则

14.1.1 精确匹配

精确匹配是指<url-pattern>中配置的值必须与 url 完全精确匹配。

```
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
    <url-pattern>/demo.do</url-pattern>
</servlet-mapping>
```

http://localhost:8888/demo/demo.do 匹配

http://localhost:8888/demo/suibian/demo.do 不匹配

14.1.2 扩展名匹配

在<url-pattern>允许使用统配符“*”作为匹配规则，“*”表示匹配任意字符。在扩展名匹配中只要扩展名相同都会被匹配和路径无关。注意，在使用扩展名匹配时在<url-pattern>中不能使用“/”，否则容器启动就会抛出异常。

```
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

http://localhost:8888/demo/abc.do 匹配

http://localhost:8888/demo/suibian/haha.do 匹配

http://localhost:8888/demo/abc 不匹配

14.1.3 路径匹配

根据请求路径进行匹配，在请求中只要包含该路径都匹配。“*”表示任意路径以及子路径。

```
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
    <url-pattern>/suibian/*</url-pattern>
</servlet-mapping>
```

http://localhost:8888/demo/suibian/haha.do 匹配

http://localhost:8888/demo/suibian/hehe/haha.do 匹配

http://localhost:8888/demo/hehe/heihei.do 不匹配

14.1.4 任意匹配

匹配“/”。匹配所有但不包含 JSP 页面

```
<url-pattern>/</url-pattern>
```

http://localhost:8888/demo/suibian.do 匹配

http://localhost:8888/demo/addUser.html 匹配

http://localhost:8888/demo/css/view.css 匹配

http://localhost:8888/demo/addUser.jsp 不匹配

http://localhost:8888/demo/user/addUser.jsp 不匹配

匹配所有

```
<url-pattern>/*</url-pattern>
```

http://localhost:8888/demo/suibian.do 匹配

http://localhost:8888/demo/addUser.html 匹配

http://localhost:8888/demo/suibian/suibian.do 匹配

14.1.5 优先顺序

当一个 url 与多个 Servlet 的匹配规则可以匹配时 则按照“ 精确路径 > 最长路径 > 扩

展名”这样的优先级匹配到对应的 Servlet。

14.1.6 考考你

Servlet1 映射到 /abc/*

Servlet2 映射到 /*

Servlet3 映射到 /abc

Servlet4 映射到 *.do

当请求 URL 为“/abc/a.html”，“/abc/*”和“/*”都匹配，Servlet 引擎将调用 Servlet1。

当请求 URL 为“/abc”时，“/abc/*”和“/abc”都匹配，Servlet 引擎将调用 Servlet3。

当请求 URL 为“/abc/a.do”时，“/abc/*”和“*.do”都匹配，Servlet 引擎将调用 Servlet1。

当请求 URL 为“/a.do”时，“/*”和“*.do”都匹配，Servlet 引擎将调用 Servlet2。

当请求 URL 为“/xxx/yyy/a.do”时，“/*”和“*.do”都匹配，Servlet 引擎将调用 Servlet2。

14.2URL 映射方式

在 web.xml 文件中支持将多个 URL 映射到一个 Servlet 中，但是相同的 URL 不能同时映射到两个 Servlet 中。

14.2.1 方式一

```
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
    <url-pattern>/suibian/*</url-pattern>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

14.2.2 方式二

```
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
```

```
    <url-pattern>/suibian/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>demoServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

15 基于注解式开发 Servlet

在 Servlet3.0 以及之后的版本中支持注解式开发 Servlet。对于 Servlet 的配置不在依赖于 web.xml 配置文件，而是使用@WebServlet 注解完成 Servlet 的配置。

15.1@WebServlet 注解中属性

属性名	类型	作用
initParams	WebInitParam[]	Servlet 的 init 参数
name	String	Servlet 的名称
urlPatterns	String[]	Servlet 的访问 URL，支持多个
value	String[]	Servlet 的访问 URL，支持多个
loadOnStartup	int	自启动 Servlet
description	String	Servlet 的描述
displayName	String	Servlet 的显示名称
asyncSupported	boolean	声明 Servlet 是否支持异步操作模式

15.2@WebInitParam 注解中的属性

属性名	类型	作用
name	String	param-name
value	String	param-value
description	String	description

16 文件上传

在 Servlet3.0 之前的版本中如果实现文件上传需要依赖 apache 的 Fileupload 组件，在 Servlet3.0 以及之后的版本中提供了 Part 对象处理文件上传，所以不在需要额外的添加 Fileupload 组件。

在 Servlet3.0 以及之后的版本中实现文件上传时必须要在 Servlet 中开启多参数配置：

web.xml

```
<multipart-config>
    <file-size-threshold></file-size-threshold>
    <location></location>
    <max-file-size></max-file-size>
    <max-request-size></max-request-size>
</multipart-config>
```

元素名	类型	描述
<file-size-threshold>	int	当数据量大于该值时，内容将被写入临时文件。
<location>	String	存放生成的临时文件地址
<max-file-size>	long	允许上传的文件最大值（byte）。默认值为 -1，表示没有限制
<max-request-size>	long	一个 multipart/form-data 请求能携带的最大字节数（byte），默认值为 -1，表示没有限制。

注解

@MultipartConfig

属性名	类型	描述
fileSizeThreshold	int	当数据量大于该值时，内容将被写入临时文件。
location	String	存放生临时成的文件地址
maxFileSize	long	允许上传的文件最大值（byte）。默认值为 -1，表示没有限制
maxRequestSize	long	一个 multipart/form-data 请求能携带的最大字节数（byte），默认值为 -1，表示没有限制。

Part 对象中常用的方法

String getContentType()

获取上传文件的文件的 MIME 类型

long getSize()

上传文件的大小

String getSubmittedFileName()

上传文件的原始文件名

String getName()

获取<input name="upload" ...>标签中 name 属性值

String getHeader(String name)

获取请求头部

Collection<String> getHeaderNames()

获取所有请求头部名称

InputStream getInputStream()

获取上传文件的输入流

void write(String path)

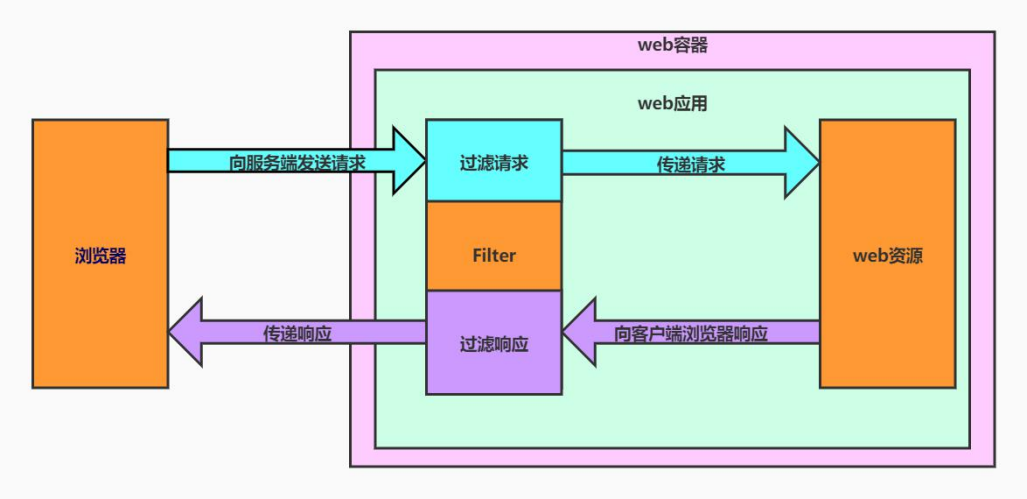
保存文件至服务器

17 Filter 过滤器

Filter 过滤器是 Servlet 中的一个组件。并不是 JavaEE 平台中的技术规范。

17.1过滤器作用

对从客户端向服务器端发送的请求进行过滤，也可以对服务器端返回的响应进行处理。



17.2Filter 的使用

17.2.1 Filter 对象的创建

创建一个 Class 实现 Filter 接口，并实现接口中三个抽象方法。

init()方法：初始化方法，在创建 Filter 后立即调用。可用于完成初始化动作。

doFilter()方法：拦截请求与响应方法，可用于对请求和响应实现预处理。

destroy()方法：销毁方法，在销毁 Filter 之前自动调用。可用于完成资源释放等动作。

17.2.2 在 Filter 中设置请求编码

需求：在 Filter 中实现对请求的编码的设置。

17.2.3 FilterConfig 对象的使用

FilterConfig 对象是用来读取<filter>中<init-param>初始化参数的对象。该对象通过参数传递到 init 方法中，用于读取初始化参数。

filterConfig.getInitParameter("name")

通过 name 获取对应的 value。

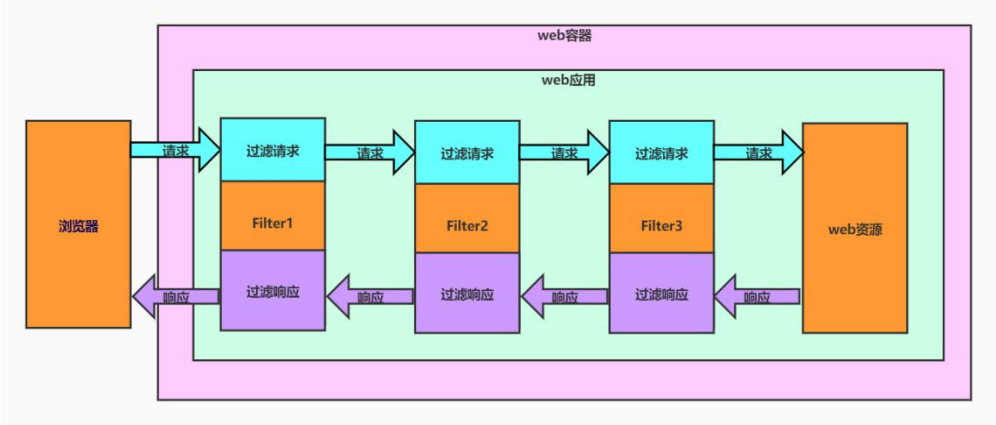
filterConfig.getInitParameterNames()

返回该 Filter 中所有<param-name>中的值。

17.3FilterChain（过滤器链）

Filter 技术的特点是在对请求或响应做预处理时，可实现“插拔式”的程序设计。我们

可以根据自己需求添加多个 Filter，也可以根据需求去掉某个 Filter，通过修改 web.xml 文件即可实现。那么如果有多个过滤器对某个请求及响应进行过滤，那么这组过滤器就称为过滤器链。过滤器的执行顺序和<url-pattern>有关。



Filter 执行顺序

则按照在 web.xml 文件中配置的上下顺序来决定先后。在上的先执行，在下的后执行。

17.4Filter 的生命周期

Filter 的生命周期是由容器管理的。当容器启动时会实例化 Filter 并调用 init 方法完成初始化动作。当客户端浏览器发送请求时，容器会启动一个新的线程来处理请求，如果请求的 URI 能够被过滤器所匹配，那么则先调用过滤器中 的 doFilter 方法，再根据是否有 chain.doFilter 的指令，决定是否继续请求目标资源。当容器关闭时会销毁 Filter 对象，在销毁之前会调用 destroy 方法。

17.5基于注解式开发 Filter

Filter 支持注解式开发，通过@WebFilter 注解替代 web.xml 中 Filter 的配置。

属性名	类型	作用

filterName	String	指定过滤器的 name 属性
urlPatterns	String[]	拦截请求的 URL，支持多个
value	String[]	拦截请求的 URL，支持多个
description	String	过滤器的描述
displayName	String	过滤器的显示名称
initParams	WebInitParam[]	指定一组过滤器初始化参数，等价于 <init-param> 标签。

使用注解式开发 Filter 时，执行顺序会根据 Filter 的名称进行排序的结果决定调用的顺序。

18 Listener 监听器

监听器用于监听 web 应用中某些对象的创建、销毁、增加，修改，删除等动作的发生，然后作出相应的响应处理。当范围对象的状态发生变化的时候，服务器会自动调用监听器对象中的方法。

18.1监听器分类

按监听的对象划分，可以分为：

- ServletContext 对象生命周期监听器与属性操作监听器
- HttpSession 对象生命周期监听器与属性操作监听器
- ServletRequest 对象生命周期监听器与属性操作监听器

18.2ServletContext 监听器

18.2.1 ServletContext 对象的生命周期监听器

ServletContextListener 接口定义了 ServletContext 对象生命周期的监听行为。

void contextInitialized(ServletContextEvent sce)

ServletContext 对象创建后会触发该监听方法，并将 ServletContext 对象传递到该方法中。

void contextDestroyed(ServletContextEvent sce)

ServletContext 对象在销毁之前会触发该监听方法，并将 ServletContext 对象传递到该方法中。

18.2.2 ServletContext 对象的属性操作监听器

ServletContextAttributeListener 接口定义了对于 ServletContext 对象属性操作的监听行为。

void attributeAdded(ServletContextAttributeEvent scae)

向 ServletContext 对象中添加属性时会触发该监听方法 ,并将 ServletContext 对象传递到该方法中。触发事件的方法 servletContext.setAttribute("key","value")。

void attributeRemoved(ServletContextAttributeEvent scae)

当从 ServletContext 对象中删除属性时会触发该监听方法 ,并将 ServletContext 对象传递到该方法中。触发事件方法 servletContext.removeAttribute("key")。

void attributeReplaced(ServletContextAttributeEvent scae)

当从 ServletContext 对象中属性的值发生替换时会触发该监听方法，并将 ServletContext 对象传递到该方法中。触发事件的方法 servletContext.setAttribute("key","value")。

18.3HttpSession 监听器

18.3.1 HttpSession 对象的生命周期监听器

HttpSessionListener 接口定义了 HttpSession 对象生命周期的监听行为。

void sessionCreated(HttpSessionEvent se)

HttpSession 对象创建后会触发该监听方法，并将已创建 HttpSession 对象传递到该方法

中。

void sessionDestroyed(HttpSessionEvent se)

HttpSession 对象在销毁之前会触发该监听方法，并将要销毁的 HttpSession 对象传递到该方法中。

18.3.2 HttpSession 对象的属性操作监听器

HttpSessionAttributeListener 接口定义了对于 HttpSession 对象属性操作的监听行为。

void attributeAdded(HttpSessionBindingEvent se)

向 HttpSession 对象中添加属性时会触发该监听方法，并将 HttpSession 对象传递到该方法中。触发事件的方法 HttpSession.setAttribute("key","value")。

void attributeRemoved(HttpSessionBindingEvent se)

当从 HttpSession 对象中删除属性时会触发该监听方法，并将 HttpSession 对象传递到该方法中。触发事件方法 HttpSession.removeAttribute("key")。

void attributeReplaced(HttpSessionBindingEvent se)

当从 HttpSession 对象中属性的值发生替换时会触发该监听方法，并将 HttpSession 对象传递到该方法中。触发事件的方法 HttpSession.setAttribute("key","value")。

18.4HttpServletRequest 监听器

18.4.1 HttpServletRequest 对象的生命周期监听器

ServletRequestListener 接口定义了 ServletReqest(是 HttpServletRequest 接口的父接口类型)对象生命周期的监听行为。

void requestInitialized(ServletRequestEvent sre)

HttpServletRequest 对象创建后会触发该监听方法，并将已创建 ServletRequest 对象传递到该方法中。

void requestDestroyed(ServletRequestEvent sre)

HttpServletRequest 对象在销毁之前会触发该监听方法，并将要销毁的 ServletRequest 对象传递到该方法中。

18.4.2 HttpServletRequest 对象的属性操作监听器

ServletRequestAttributeListener 接口定义了对于 HttpServletRequest 对象属性操作的监听行为。

void attributeAdded(ServletRequestAttributeEvent srae)

向 HttpServletRequest 对象中添加属性时会触发该监听方法，并将 ServletRequest 对象传递到该方法中。触发事件的方法 HttpServletRequest.setAttribute("key","value")。

void attributeRemoved(ServletRequestAttributeEvent srae)

当从 HttpServletRequest 对象中删除属性时会触发该监听方法，并将 ServletRequest 对象传递到该方法中。触发事件方法 HttpServletRequest.removeAttribute("key")。

void attributeReplaced(ServletRequestAttributeEvent srae)

当从 HttpServletRequest 对象中属性的值发生替换时会触发该监听方法，并将 ServletRequest 对象传递到该方法中。

触发事件的方法 HttpServletRequest.setAttribute("key","value")。

18.5基于注解式开发监听器

Listener 支持注解式开发，通过@WebListener 注解替代 web.xml 中 Listener 的配置。

19 Filter 与 Listener 设计模式

“知其然知其所以然”。

19.1Filter 的设计模式

在 Servlet 的 Filter 中使用的责任链设计模式。

19.1.1 责任链模式特点

责任链（Chain of Responsibility）：责任链模式也叫职责链模式，是一种对象行为模式。在责任链模式里，很多对象由每一个对象对其下一个对象的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不需要知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

19.1.2 责任链的优缺点

优点：

- 降低了对象之间的耦合度。
- 增强了系统的可扩展性。
- 增强了给对象指派职责的灵活性。
- 责任链简化了对象之间的连接。
- 责任分担。每个类只需要处理自己该处理的工作。

缺点：

- 不能保证请求一定被接收。
- 对比较长的责任链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
- 可能会由于责任链的错误设置而导致系统出错，如可能会造成循环调用。

19.2Listener 的设计模式

在 Servlet 的 Listener 中使用的观察者设计模式。

19.2.1 观察者模式的特点

观察者模式（Observer Pattern）：观察者模式是一种对象行为模式。它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

19.2.2 观察者模式的优缺点

优点：

- 观察者和被观察者是抽象耦合的。
- 建立一套触发机制。

缺点：

- 如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

- 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。